*Research Inventy: International Journal of Engineering And Science*
*Vol.4, Issue 7 (July 2014), PP 54-64*
*Issn (e): 2278-4721, Issn (p):2319-6483, www.researchinventy.com*

# A Regression Test Case Selection and Prioritization for Object-Oriented Programs using Dependency Graph and Genetic Algorithm

Samaila Musa

**ABSTRACT:** *Regression testing is very important activity in software testing. The re-execution of all test cases during regression testing will be costly. The effective and efficient test case selection from the existing test suite becomes very critical issue in regression testing. This paper presents an evolutionary regression test case prioritization for object-oriented software based on dependence graph model analysis of the affected program using Genetic Algorithm. The approach is based on optimization of selected test case from test suite T. The goal is to identify changes in a method's body due to data dependence, control dependence and dependent due to object relation such as inheritance and polymorphism, select the test cases based on affected statements and ordered them based on their fitness by using GA. The number of affected statements determined how fit a test case is good for regression testing. A case study was reported to provide evidence of the feasibility of the approach and its benefits in increasing the rate of fault detection and reduction in regression testing effort. The goodness of this ordering is measured using Average Percentage of rate of Faults Detection (APFD) metric to evaluate the effectiveness and efficiency of the approach. It was observed that our proposed approach is more efficient and effective in regression testing.*

**KEYWORDS***: Regression testing, regression test case, evolutionary algorithm, genetic algorithm, regression test case prioritization, and system dependence graph.*

## I. INTRODUCTION

Software maintenance activity is an expensive phase account for nearly 60% of the total cost of the software production [17]. Regression testing is an important phase in software maintenance activity to ensure that modifications due to debugging or improvement do not affect the existing functionalities and the initial requirement of the design [18] and it almost takes 80% of the overall testing budget and up to 50% of the cost of software maintenance [19]. Regression testing is a software testing activity normally conducted after software is changed, and its helps not only to ensure that changes due to debugging or improvement do not affect the existing functionalities but also the changes do not affect the initial requirement of the design. Regression test selection is an activity that select test cases from an existing test suite, that need to be rerun to ensure that modified parts behave as intended and the modification have not introduce sudden faults.Regression test selection is a way that test cases are selected from an existing test suite, that need to be rerun to ensure that modified parts behave as intended and the modification have not introduce sudden faults. Prioritization of test cases to be used in testing modified program means reduction in the cost associated with regression testing.

Identifying test cases that exercised modified parts of the software is the main objective of regression test selection. The challenge in regression testing is the prioritization of selected test cases by identifying and selecting of best test cases from the selected test cases, and selecting good test cases will reduce execution time and maximize the coverage of fault detection. Regression test selection technique will help in selecting a subset of test cases from the test suite. The easiest way in regression testing is the tester simply executes all of the existing test cases to ensure that the new changes are harmless and is referred as retest-all method [10]. It is the safest technique, but it is possible only if the test suite is small in size. The test case can be selected at random to reduce the size of the test suite. But most of the test cases selected randomly can result in checking small parts of the modified software, or may not even have any relation with the modified program. Regression test selection techniques will be an alternative approach.

Problem definition:

Let P be a certified program tested with test suite T, and P' be a modified program of P. During regression testing of P', T and information about the testing of P with T are available for use in testing P'.

To solve the above problem, Rothermel and Harrold [19] have outlined a typical selective retest technique that:

- Identify changes made to P by creating a mapping of the changes between P and P'
- Use the result of the above step to select a set T' subset of T that may reveal changes-related faults in P'
- Use T' to test P', to establish the correctness of P` with respect to T`
- Identify if any parts of the system have not been tested adequately and generate a new set of test case T''.
- Use T'' to test P`, to establish the correctness of P` with respect to T``
- Create T```, a new test suite and test history for P`, from T, T`, and T``.

Regression testing approach can be based on source code, i.e., code-based and based on design, i.e., design-based, many of them were proposed by the researchers. The more safe and easy to make are the approaches that generate the model directly from the source code of the software.Researchers have proposed many code-based approaches [9,19,20] by identifying modifications in the level of source code. Other researchers [1,2,3,6,7,8,10,12] address the issues of object-oriented programming, but there is need for further research to consider the basic concept of object-oriented features (such as inheritance, polymorphism, etc.,) as a bases in identifying changes. Researchers have proposed various approaches [5,13,14,15,16] to address the issues related to prioritization using heuristic approaches that may not produce optimal solutions. An approach for test case prioritization was presented in [13,14] based on analysis of dependence model. The authors constructed a dependence model of a program from its source code, and when the program is modified, the model is updated to reflect the changes. The affected nodes are determined by performing forward slices using the changed nodes as slicing criterion. The test cases that covered the affected nodes are selected for regression testing. The selected test cases are then prioritized by assigning initial weights. The weights are used as bases for prioritization, which may result in selection of test case that is not much relevant, which will result in increase of regression testing time. In [15], a requirement based system level test case prioritization was proposed in order to reveal more severe faults at an earlier stage based on factor oriented in regression testing using GA (PFRevSevere) was proposed. An approach was proposed in [5,16] that prioritized based on rate of faults detected and impact of the faults. A test suite reduction approach was proposed in [6] to select a subset of test cases that executes the changed requirements. The performance of the proposed approach was evaluated using percentage of requirement coverage. The results shown an improvement compared with the state-of-art.

In this paper we present an evolutionary prioritized approach that will select best test cases from existing test suite T used to test the original program P by using Dependence Graph [11] as an intermediate to identify the changes in P, at statements level. Identification of changes using this kind of graph will leads to précised detection of changes. The changed statements will be used to identify affected statements, and test cases that execute the affected statements are selected for regression testing. The selected test cases will be prioritized by using genetic algorithm in order to have a superior rate of fault detection. This approach will reduce the cost of regression testing by increasing the rate of faults detection and reducing the number of test cases to be used in testing the modified program.The rest of this paper is organized as follows. In the next section, we describe Extended System Dependence Graph (ESDG). Section 3 introduced GA. Section 4 describes Average percentage of rate of Faults Detection (APFD). In section 5, we present our test cases selection and prioritization technique. Section 6 presents the results and discussion. We conclude this paper at section 7.

## II. EXTENDED SYSTEM DEPENDENCE GRAPH

In this section, we describe the dependency graph based on the approach presented in [11]. ESDG was used to model object-oriented programs and is an extension of System Dependence Graph (SDG) [8] used to model procedural programs.

Extended System Dependence Graph [11] is a graph that can represents control and data dependencies, and information pertaining to various types of dependencies arising from object-relations such as association, inheritance and polymorphism. Analysis at statement levels with ESDG model helps in identifying changes at basic simple statement levels, simple method call statements, and polymorphic method calls.

ESDG is a directed, connected graph G = (V, E), that consist of set of V vertices and a set E of edges.

**ESDG Vertices :** A vertex v represents one of the four types of vertices, namely, statement vertices, entry vertices, parameter, and polymorphic vertices.

**Statement Vertex :** Is used to represent program statements present in the methods body.

**Parameter Vertex :**This is used to represent parameter passing between a caller and callee method. They are of four types: formal-in, formal-out, actual-in, and actual-out. Actual–in and actual-out vertices are created for each call vertex and create formal-in and formal-out vertices for each method entry vertex.
**Entry Vertex :** Methods and classes have entry vertices use to represent the entry of method and class respectively.

**Polymorphic Choice Vertex :** It is used to represent dynamic choice among the possible bindings in a polymorphic call.

**ESDG Edges :** An edge e represent one of the six edges, namely, control dependence edges, data dependence edges, parameter dependence edges, method call edges, summary edges, and class member edges.

**Control Dependence Edge :**It is used to represents control dependence relations between two statement vertices.

**Data Dependence** Edge :It is used to represents data dependence relations between statement vertices.

**Call Edge :** It is used to connect a calling statement to a method entry vertex. It also connects various possible polymorphic method call vertices to a polymorphic choice vertex.

**Parameter Dependence Edge :** It is used for passing values between actual and formal parameters in a method call. It is of two types: parameter-in and parameter-out edges.

**Summary Edge :**It is used to represents the transitive dependence between actual-in and actual-out vertices.

**Class Member Edge :** It is used to represents the membership relation between a class and its methods.

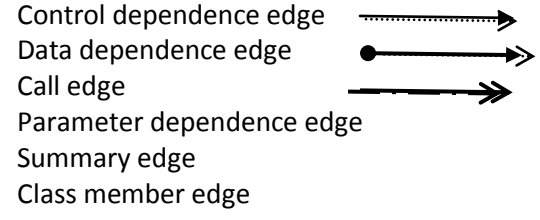
Figure 1a represents the different graphical symbols used to represent the different types of edges, Figure 1b represents a simple class that computes the sum of numbers 1 to 9, and Figure 1c represents partial ESDG of the codes in Figure 1b.

## III. GENETIC ALGORITHM

Many real life problems have been solved using evolutionary algorithms, and GA is one such evolutionary algorithm. Some of the real life problems where GA was applied are:

- For railway scheduling problem
- The travelling salesman problem
- The vehicle routing problem and
- Many field of software engineering.

Genetic Algorithm has emerged as optimization technique and search method. Problems being solved by GA are represented by a population of chromosomes as the solution to the problems. A chromosome can be string of binary digits, integer, real or characters, and each string that makes up a chromosome is called a gene. This initial population can be totally random or can be created manually using processes such as heuristic technique.




)hical symbols of edges

```
(
S2          public static int i;
S3          public static int sum;
E4          public void TSum( ) {   // constructor
S5                sum =0;
S6                i = 1;
          }
 E7         public void calculate( ) {
```

```
S8              while (i<10) {
S9              sum = add(sum, i);
S10             i = add(i, 1);
                }
S11             System.out.println("sum = " + sum);
S12             System.out.println ("i = " + i);
            }
E13         static int add (int a, int b) {
S14           return(a+b);
            }
        }
```

Figure 1b. A simple class

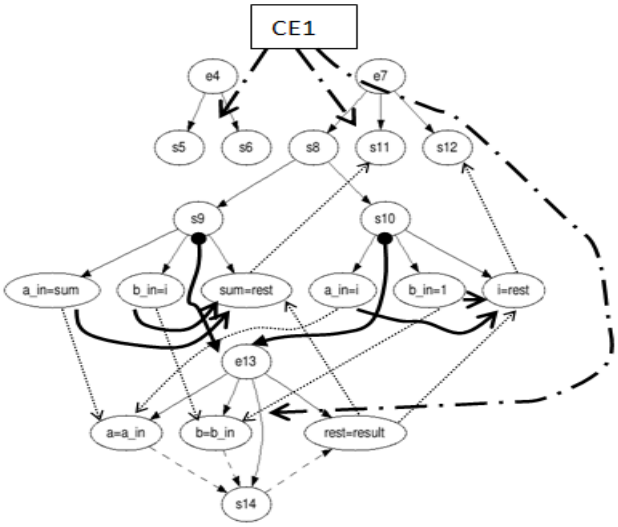



Figure 1b. Partial ESDG of class of figure 1b.

## IV. AVERAGE PERCENTAGE OF RATE OF FAULTS OF DETECTION (APFD)

To evaluate the performance of regression test case prioritization, researchers [5,13,14,15,16] used APFD metric. The APFD metric as widely used in gauging the performance of program P and test suite T is given as:

$$APFD(T,P) = 1 - \frac{(Tf_1+Tf_2+Tf_3+ ..+Tf_m)}{nm} + \frac{1}{2n} \qquad (1)$$

Where,
m is the number of faults
n is the number of test cases
$Tf_1+Tf_2+Tf_3+ ……..+Tf_m$ are the position of the first test in T that expose the faults 1,2, ….,m.

## V. PROPOSED REGRESSION TEST CASE SELECTION & PRIORITIZATION FRAMEWORK

This paper presents an approach for the selection of test cases T` from the test suite T to be used in testing the modified program P`, and prioritized the selected test cases in order that will increases their rate of detecting faults.

Figure 2, illustrate the various activities of our proposed test case selection and prioritization framework.

**Identify Changes :** The changes between *P* and the modified program *P`* are identified in this step, via semantic analysis of the source code of the software. A file named "*changed"* is used to store the identified statement level differences. This is shown in Figure 1 by the result of identify changes phase.

The scopes of the changes in our approach are addition and deletion of object.

**Adding Object** : Adding object in ESDG can be identified by Identify changes phase. Adding of object in object-oriented programming can be addition of method call statements, or simple statements such as conditionals, loops and assignment statements in the program. Figure 3ai and Figure 3aii represent program P and its modified version P` codes, and Figure 3bi and Figure 3bii represent the corresponding ESDG and its updated ESDG of simple statements addition respectively.In Figure 3ai, statements (vertices) S2, S3, S4 and S5

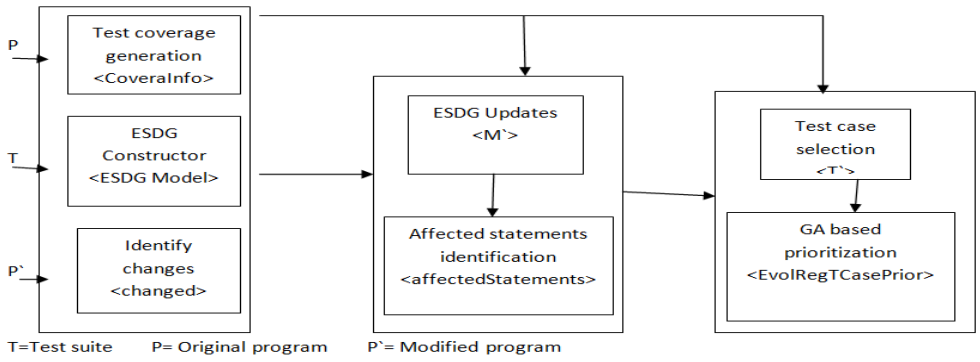
are control dependence on E1 (method entry vertex), vertices S3, S4 and S5 are data dependent on S2, and S5 is data dependent on S2, S3 and S4 as shown in Figure 3bi. In Figure 3aii, statement S4 and S5 are not data dependent on S2, but are on the added statement S3a. The added statement S3a is data **dependent on S2 as shown in Figure 3bii. Statement S3a is identify as the changes between P and P`, and is saved as changed node.**

```
E1 void m1() {
S2   int x = 1;
S3   int y = x + 2;
S4   int p = x * 6;
S5  System.out.print(x, y, p);
  }
```

Figure 3ai. A sample method codes

```
E1 void m1() {
S2    int x = 1;
S3    int y = x + 2;
S3a   x = x+1; // added statement
S4     int p = x * 6;
S5     System.out.print (x, y, p);
  }
```

Figure 3aii. Modified codes of simple addition



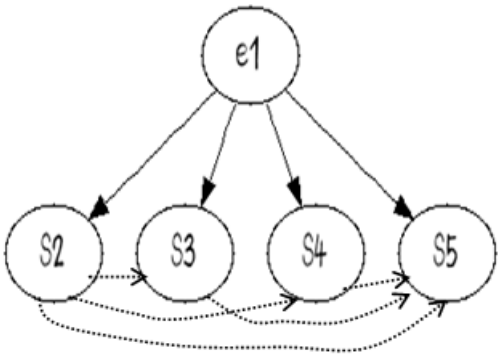




Figure 3bi. ESDG of fig. 3ai.

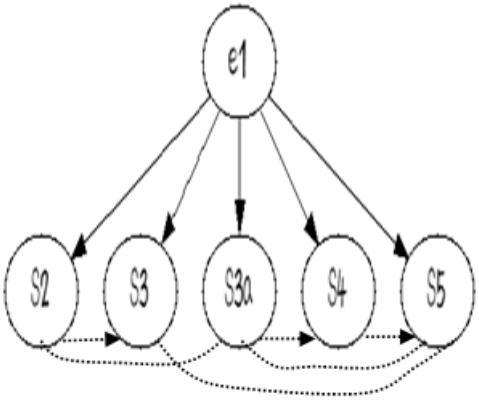



Figure 3bii. Updated ESDG of fig. 3bi.

```
CE1      class A {
E2           Public int x, y;
E3           void A () {
S4               x = 5;
S5               y = 7;  }
E6           void increment (y) {
S6a        sum (y, 1);  }
   // added method call statement
E7           void sum (int x, int y) {
S8               x += y;
                 }
          } // class
```

Figure 4a. Addition of a simple method call

**Deleting of Object:**An example of deletion of simple statement is presented in Figure 5a, and in the case of deletion of method call statement, we used the code and ESDG in Figure 4a and Figure 4b respectively. In Figure 5b, the deleted node is S4 marked by dash line. The statement vertices S5 and S6 are data dependent on S4, so before deleting S4, nodes S5 and S6 are identified changes. The edges from deleted node to nodes S5 and S6 are deleted and saved the identified nodes as changed nodes, and also a data dependence edge from node S2 to the deleted node is removed due to deletion of S4.From Figure 4a, we assume the deleted method call statement is S6a. To update the model by deleting the node S6a in ESDG, first identify the changed nodes, i.e., nodes that are control dependent or data dependent or dependent due to object relation such as inheritance and polymorphism, and saved these nodes in the file named "changed" to be used later. Secondly, there is need to remove all the parameter edges, the simple call edge, and control dependence edge. Then the vertices are deleted.

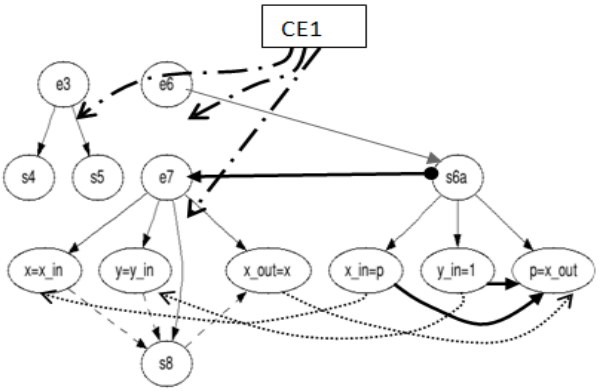



Figure 4b. Updated ESDG for fig. 4a.

```
E1 void m1() {
S2   int x = 1;
S3   int y = x + 2;
```

```
S4   x = x+1; // deleted stat
S5   int p = x * 6;
S6  System.out.print(x, y, p);
  }
```

Figure 5a. Deletion of simple statement

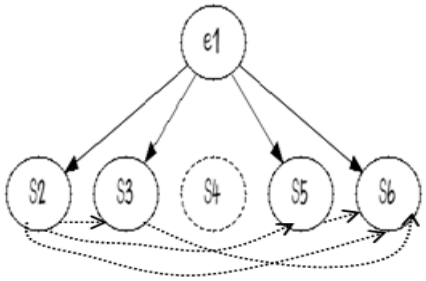



Figure 5b. ESDG for deletion of simple statement

**Test Coverage Generation :** Program P is instrumented at the statement levels. The code statements are executed with the original test suite T and to write traces for each test case in order to generate information pertaining to the specific statements that are executed for each test case. The information generated in this stage is saved in a file named *"coverageInfo*" for later use.

**ESDG Model Constructor :** ESDG model for the original program P is constructed using a technique similar to [11], and is described in section 2. Graphviz [4] is used to represents our graphs constructed.

**ESDG Model Updates :** The model constructed for P is updated using information from changed file during each regression testing to make it correspond to the modified program P` and the updated ESDG model is denoted by *M*`.

**Affected Statements Identification :**To identify the affected statements, a forward slice is constructed on the updated model M` using the information from changed file. Each change node in changed file is used as slicing criterion to determine the affected nodes in each statement, and this is performs by statements in line 10-13 of algorithm 1. The change nodes stored in changed file are used to identify the affected statements. The affected statements are statements that were affected directly by the modifications or as the result of control dependence or data dependence or dependent as a result of object relation such as inheritance and polymorphism on the affected node from the updated model M`, and denoted by"*affectedStat"*.

**Test Case Selection :** We used EvolRegTCasePrior algorithm presented in algorithm 1, to select test cases that execute the affected statements in the updated model M` for regression testing, and donated as T`. The selection is performs by statements in line 14-21.

Given the following set of test suite

T = {t1 t2 t3 t4 t5 t6 t7 t8 t9}

We used test coverage generator to get the following coverage information of the test cases and save as *"coverageInfo"*:

t1 = {n1 n2 n6},
t2 = {n2 n7 n9 n10},
t3 = {n3 n7 n8},
t4 = {n4 n6 n7},
t5= {n3 n4 n5 n9 n10},
t6 = {n3 n4 n6 n8 n10},
t7 = {n6 n7 n8},
t8 = {n8 n9 n10}.

The faults are:

n1 n2 n3 n4 n5 n6, identified by performing forward slicing using "changed nodes" as slice criterion.

The selected test cases will be:

t1 = {n1 n2}
t2 = {n2}
t3 = {n3}

t4 = {n4}
t5 = {n3 n4 n5}
t6 = {n3 n4}
I.e., the selected test case T' will be:
T` = {t1 t2 t3 t4 t5 t6}

**Test Case Prioritization :** The selected test cases need to be ordered to increase the rate at which faults are detected while running them. Test case prioritization technique needs to be given some guides in order to improve it efficiency. We proposed an evolutionary approach using GA to optimize the selected test cases. The selected test cases T` will be prioritized using evolutionary algorithm presented in algorithm 2, denoted as *"GAPriorTCase*".

*Algorithm 1*
*EvolRegTCasePrior(M', T, CoveraInfo, Changed, AffectedNodes, T', EvaPrioTcase)*
*{*
*1 Changed: is the set of changed objects*
*2 M': is the updated extended system dependence graph*
*3 T: is the set of all test cases*
*4 Coverainfo: is the set of nodes covered by a test case*
*5 affectedNodes: is the set of affected nodes*
*// changed objects and dependent nodes*
*6 T': is the set of selected test cases*
*7 EvaPrioTcase: is the set of prioritized selected test cases*
*8 affectedNodes = { ø }*
*9 T' = { ø }*
*10 For (node n : Changed ) {*
*11 Find nodes (nDep) that are dependent on node n*
*12 affectedNodes = affectedNodes U nDep*
*13 }*
*14 While (affectedNodes != Null) DO*
*15 {*
*16 For ( node a : affectedNodes)*
*17 {*
*18 Find all test cases that cover node a (tcase)*
*19 T' = T' U tcase*
*20 }*
*21 }*
*22 GAPriorTCase(CoveraInfo, T', EvaPrioTcase)*
*23 }*

*Algorithm 2*
*GAPriorTCase(CoveraInfo, T', EvaPrioTcase)*
*{*
*1 Encode the selected test cases (T')*
*2 T' = { $i_1$, $i_2$, ... ....$i_p$)*
*3 // where $i_1$, $i_2$, ……. $i_p$ are the positions of test cases*
*// t1, t2, ….. tp in the test suite T*
*4 Generate two populations P1 and P2 from T' to be the initial population*
*5 P1 is the set T' and*
*6 p2 is the reversed order of T'*
*7 Evaluate the fitness of each parent using:*
*8 $Ft(pi) = \sum n(t_j) * p_{ij}$ for j=1 to k*
*9 REPEAT {*
*10 child = ordered crossover (P1, P2)*
*11 child = swap mutation ( child )*
*12 evaluate the fitness of child*
*13 $Ft(pi) = \sum n(tj) * pij$ for j=1 to k*
*14 add the child to the population*
*15 remove the worst chromosome from the population*

*16 } UNTIL maximum number of iterations*
*17 return the best chromosome (EvaPrioTcase)*
*18 }*

## VI. RESULT AND DISCUSSION

The empirical procedure for our proposed approach is: design a test suite (T) for a program using white box testing technique, generate and save the coverage information for each test case, construct extended system dependency graph for the program, introduce changes into the program, reflect the changes in extended system dependence graph, get all the affected statements/nodes that are dependent on the changes; the affected statements are statements that were affected directly by the modifications or affected as the result of control dependence or data dependence or dependent as a result of object relation such as inheritance and polymorphism, determine which test cases in T are modification-revealing with respect to the affected statements, prioritize the selected test cases using GA, and compute the APFD of the prioritized test cases using equation (1).We used a sample test suite T used in PFRevSevere [15] with eight number of test cases, i.e., {t1,t2,t3,t4,t5,t6,t7,t8}, and five number of faults, i.e., {f1,f2,f3,f4,f5}. Our proposed approach EvolRegTCasePrior selects and prioritized the test cases as: {t5,t1,t6,t3,t2,t4}, and their approach PFRevSevere [15] prioritized the test cases as: {t5,t6,t1,t4,t2,t3,t7,t8}. The test cases and the faults detected by each test case are presented in Table 1.

Table 1. Faults detected by test cases

| Test case / Faults | T1 | T2 | T3 | T4 | T5 | T6 | T7 | T8 |
|---|---|---|---|---|---|---|---|---|
| F1 | X | | | | | | | |
| F2 | X | X | | | | | | |
| F3 | | | X | | X | X | | |
| F4 | | | | X | X | X | | |
| F5 | | | | | X | | | |

The APFD value of our approach using equation (1) with test sequence {t5 t1 t6 t3 t2 t4} is:

$$APFD(T,P) = \frac{1 - (2+2+1+1+1)}{5 * 8} + \frac{1}{2*8} = 0.8875$$

The APFD value for [18] using equation (1) with test sequence {t5 t6 t1 t4 t2 t3 t7 t8} is:

$$APFD(T,P) = \frac{1 - (3+3+1+1+1)}{5 * 8} + \frac{1}{2*8} = 0.8375$$

The above results of APFD are presented in Figure 6 for easy identification.

The comparison is between our proposed approach and an approach [15] (PFRevSevere) presented in the literature. From Table I and Figure 6, it is observed that our approach identify the faults at early stage. The APFD is better in our approach when compared to PFRevSevere. From the results, it is also observed that our proposed approach EvolRegTCasePrior is more effective than PFRevSevere in term of rate of faults detection when all test costs and faults severities are uniform.In Figure 7, we plot percentage of test cases executed against percentage of faults detected. From the result, we identified that our approach needs less test cases to reveal all faults compared to PFRevSevere [15]. It is observed that our approach EvolRegTCasePrior needs 20% of the test cases to find out all the faults. But in PFRevSevere [15], there is need of 30% of the test cases to find out all the faults.

From Figure 7, we observed that our approach EvolRegTCasePrior order of test cases detect all the faults in early stage when compared to technique proposed in PFRevSevere [15]. So, our proposed approach of test case prioritization process will reduce the regression testing time.

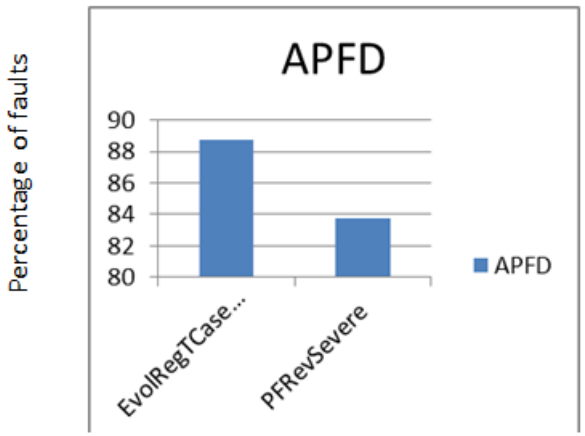



Figure 6. APFD for the approaches

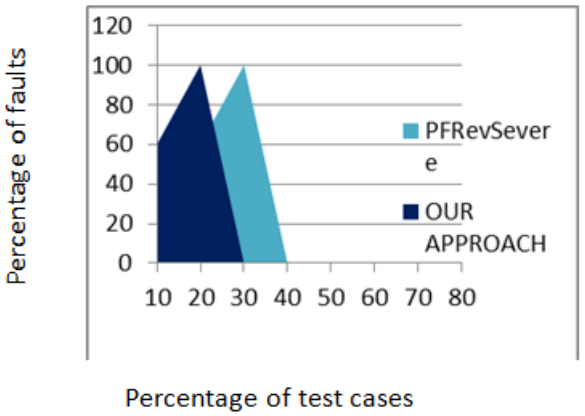



Figure 7. Percentages of test cases to needed to cover all faults

## VII. CONCLUSION

A regression test case selection and prioritization is proposed in our approach that ordered selected test cases T` from test suite T that will be used for rerun in regression testing. The approach used extended system dependence graph to identify changes at statement level of source code, store the changes in a file named changed, and generate coverage information for each test case from the source code. The changed information are used to identify the affected statements, and test cases are identify that will be rerun in regression testing based on the affected statements. The selected test cases will be prioritized using genetic algorithm in order to increases their rate of faults detection. The technique cover the different important issues that regression testing strategies need to address: change identification, test selection, test execution and test suite maintenance.The effectiveness and efficiency of the proposed approach was evaluated using APFD metric. In this paper, a sample project was used from the literature. The proposed approach provides the better results for rate of faults detection. From the results we observed that the proposed approach EvolRegTCasePrior needs only 20% of the test cases to detect all the faults, compared with the 30% needed to detect all the severe faults by PFRevSevere. Based on the measured performance obtained from the results, the proposed approach selects and prioritized test cases efficiently and effectively compared to PFRevSevere, which will result in reducing the cost of regression testing.We note that our proposed approach assumes that the ESDG are updated in a timely way, every time changes are introduced into the programs. This assumption becomes a limitation for our approach. Another limitation is that we assumed that all test costs and faults severities are uniform.The proposed framework can be used in software maintenance, especially when the changes are at statements level of object-oriented programs; such as simple statements, method calls and polymorphic calls.

## REFERENCES


[1] Beszedes A., Gergely T., Schrettner L., Jasz J., Lango L., Gyimothy T., "Code coverage-based regression test selection and prioritization in webkit," 28th IEEE International Conference on Software Maintenance, Trento, pp. 46-55, 2012.

[2] El-hamid W. S. A., El-etriby S. S., and Hadhoud M. M., "Regression Test selection technique for multi-programming language," Faculty of Computer and Information, Menofia University, Shebin-Elkom, 32511, Egypt 2009.

[3] Frechette N., Badri L., and Badri M., "Regression Test reduction for object-oriented software: A control call graph based technique and associated tool," International Scholarly Research Network Software engineering (ISRN Software Engineering), vol. 2013, ID 420394, pp. 1-10, 2013.

[4] Graphviz, http://www.graphviz.org/, last visited: March 21, 2014.

[5] Gupta R., and Yadav A. K., "Study of Test Case Prioritization Technique Using APFD," International Journal of Computers & Technology, vol. 10, no. 3, pp. 1475-1481, Aug-2013.

[6] Harris P., and Raju N., "A greedy approach for coverage-based test suite reduction," IAJIT, vol. 12, no. 1, 2013.

[7] Harrold M.J., Jones J.A., Li T., Liang D., Orso A., Pennings M., Sinha S., Spoon S.A., and Gujarathi A., "Regression test selection for java software," ACM, vol. 36, no. 11, pp. 312-326, 2001.

[8] Horwitz S., Reps T., and Binkley D., "Interprocedural slicing using dependence graphs," ACM Transactions on Programming Languages and Systems, vol. 12, no. 1, pp. 26–60, 1990.

[9] Jang Y., Munro M., and Kwon Y.R., "An improved method of selecting regression tests for C++ programs," Journal of Software Maintenance: Research and Practice, vol. 13, no. 5, pp. 331–350, 2001.

[10] Jin W., Orso A., and Xie T., "Automated Behavioral regression testing," Third International Conference on Software Testing, Verification and Validation, Washington DC, USA, pp. 137-146, 2010.

[11] Larsen L., and Harrold M. J., "Slicing object-oriented software," In Proceedings of 18th IEEE International Conference on Software Engineering, Berlin, pp. 495-505, 1996.

[12] Li Z., Harman M., and Hierons R. M., "Search algorithms for regression test case prioritization," IEEE Trans. On Software Engineering, vol. 33, no. 4, pp. 225-237, April 2007.

[13] Panigrahi C., and Mall R., "An approach to prioritize regression test cases of object-oriented programs". JCSI Trans on ICT (Springer), doi:10.1007/s40012-013-0011-7, pp. 1-15, 2013.

[14] Panigrahi C., and Mall R., **"**A heuristic-based regression test case prioritization approach for object-oriented programs," Innovations in Systems and Software Engineering (Springer), doi: 10.1007/s11334-013-0221-z, pp. 1-9, 2013.

[15] Shanmugam R., and Uma G.V., "Factors Oriented test case prioritization Technique in Regression testing using Genetic Algorithm," European Journal of Scientific Research, vol. 74, no. 3, pp. 389-402, 2012.

[16] Raperia H., and Srivastava S., "An Empirical Approach for Test Case Prioritization," International Journal of Scientific & Engineering Research, vol. 4, no. 3, pp. 1-5, March 2013.

[17] Roger S.P., Software Engineering: A practitioner's Approach, 5th ed, New York, America, McGraw-Hill, Inc., 2001.

[18] Rothermel G. and Harrold M. J., "Selecting regression tests for object-oriented software," International Conference on Software Maintenance, Victoria, CA, pp. 14–25, 1994.

[19] Rothermel G., and Harrold M. J., "A safe, efficient regression test selection technique," ACM Transactions on Software Engineering and Methodology, vol. 6, no. 2, pp. 173–210, 1997.

[20] Rothermel G., Harrold M. J, and Dedhia J., "Regression test selection for C++ software," Software Testing, Verification and Reliability, vol. 10, no. 2, pp. 77–109, 2000.